A Conceptual Framework for Adaptation

Roberto Bruni, Dipartimento di Informatica, Università di Pisa
Andrea Corradini, Dipartimento di Informatica, Università di Pisa
Fabio Gadducci, Dipartimento di Informatica, Università di Pisa
Alberto Lluch Lafuente, IMT Institute for Advanced Studies Lucca
Andrea Vandin, IMT Institute for Advanced Studies Lucca

We present a white-box conceptual framework for adaptation. We called it CoDA, for COntrol Data Adaptation, since it is based on the notion of *control data*. CoDA promotes a neat separation between application and adaptation logic through a clear identification of the set of data that is relevant for the latter. The framework provides an original perspective from which we survey a representative set of approaches to adaptation ranging from programming languages and paradigms, to computational models and architectural solutions.

Categories and Subject Descriptors: F.1.1 [**Models of computation**]: Self-modifying machines

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Adaptation, Self-*, Autonomic Computing, Computational Reflection, Control Loops, Context-Oriented Programming

ACM Reference Format:
R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, A. Vandin 2013. A Conceptual Framework for Adaptation. Draft N/A, N/A, Article N/A (0), 30 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Self-adaptive systems have been widely studied in several disciplines like Biology, Engineering, Economy and Sociology. They have become a hot topic in Computer Science in the last decade as a convenient solution to the problem of mastering the complexity of modern software systems, networks and architectures. In particular, self-adaptation is considered a fundamental feature of *autonomic systems*, often realized by specialized self-* mechanisms like self-configuration, self-optimization, self-protection and self-healing, as discussed for example in [IBM Corporation 2005].

The literature includes valuable works aimed at capturing the essentials of adaptation both in the most general sense (see e.g. [Lints 2010]) and in particular fields such as software systems (see e.g. [Salehie and Tahvildari 2009; Bouchachia and Nedjah 2012; McKinley et al. 2004; Andersson et al. 2009a; Raibulet 2008]) providing in some cases very rich surveys and taxonomies. A prominent and interesting example is the taxonomy of concepts related to self-adaptation presented in [Salehie and Tahvildari 2009], whose authors remark the highly interdisciplinary nature of the studies of such systems. Indeed, just restricting to the realm of Computer Science, active research on

This work is supported by the European IP ASCENS and STREP QUANTICOL and the Italian PRIN CINA.
Author's addresses: R. Bruni, A. Corradini, F. Gadducci: Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56126 Pisa, Italy; A. Lluch Lafuente, A. Vandin: IMT Institute for Advanced Studies Lucca, Piazza San Ponziano 6, 55100 Lucca, Italy.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

self-adaptive systems is carried out in Software Engineering, Artificial Intelligence, Control Theory, and Network and Distributed Computing, among others.

Despite all these classification efforts, there is no agreement on the *conceptual* notion of adaptation, neither in general nor for software systems, and there is no widely accepted *foundational* model for it. Lofti Zadeh noticed in [Zadeh 1963] that *"it is very difficult—perhaps impossible—to find a way of characterizing in concrete terms the large variety of ways in which adaptive behavior can be realized"*. Zadeh's concerns were conceived in the field of Control Theory but as many authors agree (e.g. [Raibulet 2008; Salehie and Tahvildari 2009; Andersson et al. 2009a; Lints 2010]), they are valid in Computer Science as well. One of reasons for Zadeh's lack of hope in a concrete unifying definition of adaptation is the attempt to subsume two aspects under the same definition: the *external* manifestations of adaptive systems, and the *internal* mechanisms by which adaptation is achieved. We shall refer to the first aspect as the *black-box* perspective on adaptation, and to the second aspect as the *white-box* one. [1]

Actually, in the realm of Software Engineering there are widely accepted informal definitions, according to which a software system is called "self-adaptive" if it *"modifies its own behavior in response to changes in its operating environment"* [Oreizy et al. 1999], where such "environment" or "context" has to be understood in the widest possible way, including both the external environment and the internal state of the system itself. Typically, such changes are applied when the software system realizes that *"it is not accomplishing what the software is intended to do, or better functionality or performance is possible"* [Laddaga 1997]. Such definitions can be exploited, to a certain extent, to measure what is often called the *degree of adaptability* or *degree of adaptivity*, i.e. to estimate or predict the system robustness under some conditions. This approach can be traced back to Zadeh's proposal [Zadeh 1963], but has been later adopted by many other authors (e.g. [Mühl et al. 2007; Hölzl and Wirsing 2011]).

The problem is that almost any software system can be considered self-adaptive according to the definitions recalled above, since any realistic system can *modify its behaviour* (for example by following different branches at the same control point) as a *reaction to a change in its context of execution* (like the change of variables or parameters). Therefore such definitions, concerned with the *behavioral* or *observational* perspective only, are of difficult applicability for distinguishing (self-)adaptive systems from plain ("non-adaptive") ones. Furthermore, they are of little use for design purposes, where separation of concerns, modularization, reuse and scalability are crucial aspects.

The development and success of many emergent Computer Science paradigms is often strongly supported by the identification of key principles around which the theoretical aspects can be conveniently investigated and fully worked out. For example, in the case of distributed computing, there have been several efforts in studying the key primitives for communication, including mechanisms for passing communication means (name mobility) or entire processes (code mobility), which has led to a widely understood theory of mobile process calculi. There is unfortunately no such agreement concerning (self-)adaptation, as it is not clear what are the characterizing structural features that distinguish such systems from plain ones.

Summarizing: (i) existing definitions of adaptation (and related notions such as *adaptivity* and *adaptability*) are not always useful in pinpointing adaptive systems, even if they allow to discard many systems that certainly are not, and (ii) such definitions do sometimes focus on the issue of *how much* a system adapts to some purpose and less on the issue of *in which manner*.

[1] The black- and white-box *perspective* should not be confused with the distinction between white- and black-box component adaptation *techniques* for components as discussed e.g. in [Bosch 1999], where *black* refers to exploiting the interface of a component and *white* to exploiting its internals.

Contribution and structure of the paper[2]. The goal of this paper is to present a conceptual framework for adaptation, proposing a simple structural criterion to portray it. This framework is called CODA, COntrol Data Adaptation, and it is presented in Section 2. Our contribution is a definition of adaptation that is general enough to be applicable to most of the approaches found in the literature, in such a way that it is tightly related (and often coincident) with these alternative notions once it is instantiated to each approach. Also, we aim at a *separation of concerns* to distinguish changes of behaviour that are part of the application logic from those where they realize the adaptation logic, calling "adaptive" only those systems capable of the latter. More precisely, we propose concrete answers to basic questions like *"is a software system adaptive?"* or *"where is the adaptation logic in an adaptive system?"*. We take a *white-box* perspective that allows us to inspect, to some extent, the internal structure of a system. Moreover, we provide the designer with a criterion to specify where adaptation is located, when it is enacted and how it is realized.

The second part of the paper (Sections 3–5) is devoted to a *proof of concept*: we overview several approaches to adaptation and validate how the CODA definition of adaptation is applied to them. This part of the paper is organized according to three of the main pillars of Computer Science: *architectural* approaches (Section 3), *foundational* models (Section 4), and *programming* paradigms (Section 5). Approaches that cover more than one of such aspects are discussed only once.

It is worth remarking that it is not the programming paradigm, the architecture or the underlying foundational model what makes a system adaptive or not. For example, adaptive systems can be programmed in any language, exactly like object-oriented systems can in imperative languages, albeit with some effort. However, it is beyond the scope of this paper to discuss approaches that do not address adaptation in an explicit way, even if they might do so implicitly.

In Section 6 we overview other surveys and taxonomies which address, from different perspectives, the same aim as our work, i.e. to shed some light around the notion of adaptation in order to identify the key features of self-adaptive systems. Finally, we wrap up our considerations and discuss current and future research in Section 7.

2. WHEN IS A SOFTWARE COMPONENT ADAPTIVE?

The behavior of a software component is governed by a program and according to the traditional view (see e.g. [Wirth 1976]), a program is made of *control* (i.e. algorithms) and *data*. Of course many more sophisticated views and paradigms have been introduced in Computer Science but this very basic view of programs is sufficient for the sake of introducing our approach. Therefore, we can say that control and data are two conceptual ingredients that in presence of sufficient computing resources determine the behaviour of a component. The CODA framework requires to make explicit the fact that the behaviour of a component depends on some well identified *control data* that can be changed to *adapt* it. At this level of abstraction we are neither concerned with the structure of control data, nor with the way they influence the behaviour of the component, nor with the causes of their modification.

Our definition of adaptation is then very simple and concrete.

[2]A preliminary version of this work was presented in [Bruni et al. 2012a]. The main differences and novelties with respect to it are: (i) the identification of different forms of control data and the corresponding discussion in Sections 2 and 7; (ii) the discussions of automata-based approaches (Section 4.1) and concurrency models (Section 4.3) specifically designed for adaptive systems; (iii) the discussion of aspect- (Section 5.2) and policy-oriented paradigms (Section 5.3); and (iv) the overview and comparison with respect to similar efforts such as surveys and taxonomies for adaptation (Section 6).

Given a component with a distinguished collection of control data, adaptation *is the runtime modification of such control data.*

From this basic definition we immediately derive several others. A component is *adaptable* if it has a distinguished collection of control data that can be modified at runtime. Thus if either the control data are not identified or they cannot be modified, then the component is not adaptable. Further, a component is *adaptive* if it is adaptable and its control data are actually modified at runtime, at least in some of its executions. Moreover, a component is *self-adaptive* if it modifies its own control data at runtime.

Given the intrinsic complexity of adaptive systems, the conceptual view of CODA might look like an oversimplification. Our goal is to show that instead it enjoys two properties that we consider fundamental: concreteness and generality.

Concreteness. Any definition of adaptation should face the problem that the judgement whether a system is adaptive or not is often subjective. Indeed, one can always argue that whatever change in the behaviour the system is able to manifest is part of the application logic, and thus should not be deemed as an adaptation. From the CODA perspective, this is captured by the fact that the collection of control data of a component can be defined, at least in principle, in an arbitrary way, ranging from the empty set ("the system is not adaptable") to the collection of all the data of the program ("any data modification is an adaptation").

As a concrete example, consider the following conditional statement.

```
if the_hill_is_too_steep then
    assemble_with_others
else
    proceed_alone
end if
```

Can it be interpreted as a form of adaptation?

From a black-box perspective the answer is "it depends". Indeed, the above statement is typical of controllers for robots operating collectively as swarms and having to face environments with obstacles (see e.g. [O'Grady et al. 2010]). As some authors observe [Harvey et al. 2005] *"obstacle avoidance may count as adaptive behaviour if [...] obstacles appear rarely. [...] If the "normal" environment is [...] obstacle-rich, then avoidance becomes [...] normal behaviour rather than an adaptation"*. In sum, the above conditional statement can be a form of adaptation in some contexts but not in others.

Now, suppose that the statement is part of the software controlling a robot, and that `the_hill_is_too_steep` is just a boolean variable set according to the value returned by some sensors. If `the_hill_is_too_steep` is considered as a standard program variable which is not part of the control data, then the change of behaviour caused by a modification of its value is not considered as an adaptation in our framework. If the variable `the_hill_is_too_steep` is instead considered as part of the control data, then modifications of its value are considered to be adaptations.

Summing up, the above question (i.e.*"can it be interpreted as a form of adaptation?"*) can be answered only after a clear identification of the control data. This means that from the white-box perspective of CODA the answer is still "it depends" as it is for the black-box case. However, there is a fundamental difference: the responsibility of declaring which behaviours are part of the adaptation logic is passed from the observer of the component to its designer. Ideally, a sensible collection of control data should be chosen to enforce a separation of concerns, allowing to distinguish neatly, if possible, the activities relevant for adaptation (those that affect the control data) from those relevant for the application logic only (that should not modify the control data).

	CONTROL DATA (as-it-is)	CONTROL DATA (class)	Section
[Horn 2001]	*	*	3.1
[Cabri et al. 2011]	*	*	3.1
[Weyns et al. 2012]	*	*	3.1
[Karsai and Sztipanovits 1999]	*	*	3.2
[Popescu et al. 2012]	*	*	4.3
[Biyani and Kulkarni 2008]	adaptation coordination strategies	adaptation strategy	4.1
[Lanese et al. 2010]	adaptation rules	adaptation strategy	5.3
[Bradbury et al. 2004],	architecture	architecture	3.1
[Kramer and Magee 2009]	architecture	architecture	3.2
[Oreizy et al. 1999]	architecture	architecture	3.2
[van Renesse et al. 1998]	module stack	architecture	3.2
[Bucchiarone et al. 2010]	current workflow	architecture	3.2
[Andrade and Fiadeiro 2002]	connectors	architecture	3.2
[Biyani and Kulkarni 2008]	architecture	architecture	4.1
[Wang et al. 2009]	effector channel	architecture	4.3
[Lanese et al. 2010]	set of activities	architecture	5.3
[Pavlovic 2000]	entire programs	entire program	4.1
[Meseguer and Talcott 2002]	rewrite rules	entire program	4.2
[Gjondrekaj et al. 2012]	processes	entire program	4.3
[De Nicola et al. 2013]	processes	entire program	4.3
[Cordy et al. 2013]	features	operation mode	4.1
[Merelli et al. 2012]	regions	operation mode	4.1
[Zhao et al. 2011]	operation mode	operation mode	4.1
[Adler et al. 2007]	active configuration	operation mode	4.1
[Schaefer and Poetzsch-Heffter 2006]	active configuration	operation mode	4.1
[Bruni et al. 2013]	control proposition	operation mode	4.1
[Zhang et al. 2009]	steady state programs	operation mode	4.1
[Iftikhar and Weyns 2012]	state space zones	operation mode	4.1
[Ehrig et al. 2010]	graph rewrite rules	operation mode	4.2
[Zhang and Cheng 2006a]	base level Petri net	operation mode	4.3
[Martín et al. 2012]	adaptor processes	operation mode	4.3
[Bravetti et al. 2012]	adaptable (local) processes	operation mode	4.3
[Salvaneschi et al. 2011]	context stack	operation mode	5.1
[Greenwood and Blair. 2004]	advices	operation mode	5.2
[Khakpour et al. 2012]	policies	operation mode	5.3

Fig. 1. Summary of some the control data forms discussed.

The CODA point of view is in line with other white-box perspectives on adaptation as we discuss in Section 6.

Generality. Any definition of adaptation should be general enough to capture the essence of the most relevant approaches to adaptation proposed in the literature. The generality of CODA is witnessed by the discussion of Sections 3–5 where we overview several approaches to adaptation, pointing out for each of them the natural candidates for control data. More explicitly, the criterion that we shall use for determining such data is the following: *a system designed according to one of such approaches manifests an adaptation exactly when the corresponding control data are subject to a change*.

Adaptive systems can be realized by resorting to a variety of computational models and programming paradigms. Consequently, the nature of control data can vary considerably, in the range of all the possible ways of influencing behavior: from simple configuration parameters to a complete representation of the program in execution that can be modified at runtime.

The variety of formalisms makes it hard to compare approaches with each other, unless one manages to map them into a unifying model of computation (which is far beyond the scope of this paper). However, for the sake of a brief discussion we enrich our intuitive view of a system as made of control, control data and ordinary data, with

additional features such as the system's *architecture* (in a general sense, including the interconnection of components, communication stacks, workflows, etc.), and the *adaptation strategy* used to enact adaptation. Moreover we shall assume that the behavior of the system or component (i.e. its control) may be structured into sub-parts that we call *operation modes*.

Such simple perspective on adaptive systems help us classify the main approaches surveyed in this paper as depicted in Figure 1. Symbol "*" is used to denote generic approaches that propose reference models where control data depends on concrete instances of the approach. For the sake of completeness the table also contains the control data as-it-is and the section where the approach is discussed.

Such classification has several advantages: (i) It provides a criterion that is orthogonal to those of the surveys and taxonomies discussed in Section 6 and to the classification by research areas along which we structure Sections 3–5. (ii) It allows us to relate approaches presented independently and in different areas but sharing, essentially, the same category of control data. This is e.g. the case of the approaches based on modes of operation that have been proposed by the Software Engineering community with paradigm-oriented approaches and by the Theoretical Computer Science community with automata and process-algebraic approaches. (iii) It allows us to compare approaches apparently similar (and falling in the same section) but based on different categories of control data. For instance, in some process-algebraic approaches the control data may reside in the communication topology or in the entire program.

Note that the classification depends on the envisioned conceptual computational formalisms where we map the approaches. We have proposed a simple one to illustrate a possible way of exploiting the notion of control data for comparison purposes, but there are certainly other possible classifications.

3. ARCHITECTURAL APPROACHES TO ADAPTATION

Several contributions to the literature describe architectural approaches to autonomic computing and self-adaptive software systems. In this section we survey some of such proposals, organizing the discussion around two main themes: reference models (Section 3.1) and reconfiguration-based approaches (Section 3.2).

3.1. Reference Models for Adaptation

We review here, among others, two very influential reference models for adaptive and self-adaptive systems: MAPE-K [Horn 2001] and FORMS [Weyns et al. 2012]. Both approaches propose general guidelines for the architecture of (self-)adaptive systems, the first one based on the presence of a control loop, the second one on the use of computational reflection. The identification of control data at this level of abstraction can only be very generic, as concrete instances may realize the reference models in significantly different ways.

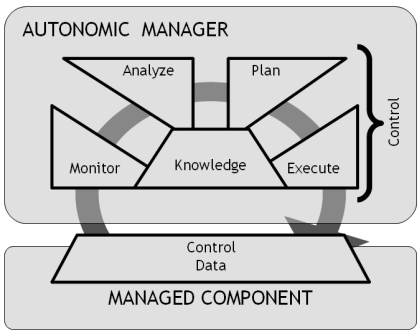

Fig. 2. Control data in MAPE-K.

The first reference models we consider is MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge), introduced in the seminal IBM paper [Horn 2001]. According to it, a self-adaptive system is made of a component implementing the application logic, equipped with a control loop that monitors the execution through suitable sensors, analyses the collected data, plans an adaptation strategy, and finally executes the adaptation of the managed component through some effectors; all the phases of the control loop access a shared knowledge

repository. The managed component is considered to be an adaptable component, and the system made of both the component and the manager implementing the control loop is considered as a self-adaptive component.

The conceptual role of the control loop induces a natural choice for the control data: these are the data of the managed component which are modified by the execute phase of the control loop. Thus the control data of a managed component is (explicitly or implicitly) available through the interface it offers to its manager, which can use it to enact its control loop, as shown in Fig. 2. Clearly, the concrete structure of control data (e.g. configuration variables, policies, programs, . . .) will depend on the specific instance of the MAPE-K model and on the computational model or programming language used to implement it, as discussed also in the next two sections.

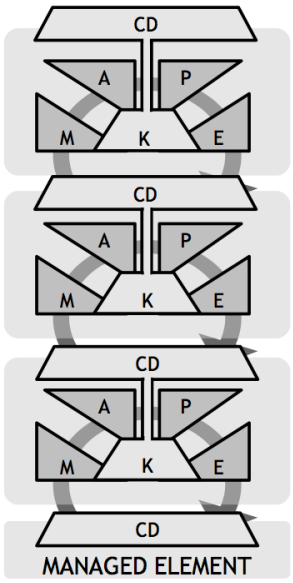

Fig. 3. Tower of adaptation.

The construction can be iterated, as the manager itself can be an adaptable component. Concrete instances of this scenario can be found, among others, in [Biyani and Kulkarni 2008; Lanese et al. 2010; Bucchiarone et al. 2011]. For example, in the latter, components follow plans to perform their tasks and re-planning is used to overcome unpredicted situations that may make current plans inefficient or impossible to realize. A component in this scenario can be adaptable, having a manager which devises new plans according to changes in the context or in the component's goals. In turn, this planning component might itself be adaptable, with another component that controls and adapts its planning strategy, for instance determining the new strategy on the basis of a tradeoff between optimality of the plans and computational cost of the planning algorithms. In this case, the planning component (that realizes the control loop of the base component) exposes some control data (conceptually part of its knowledge), thus enabling a hierarchical composition that allows building towers of adaptive components (Fig. 3).

The MAPE-K control loop is very influential in the autonomic computing community, but control loops in general have been proposed and extensively studied also by others as a key mechanism for achieving self-adaptation in software systems, also on the basis of the crucial role they play in engineering disciplines like Control Theory. An interesting survey of several types of control loops is presented in [Brun et al. 2009], which among others identifies the *Model Reference Adaptive Control loop*, where the control loop is fed with a model of the controlled component, and the *Model Identification Adaptive Control loop*, where the control loop tries to infer such a model directly from the behaviour of the component.

Typical control loop patterns are also proposed in [Cabri et al. 2011], which presents a taxonomy of design patterns for adaptation (see Fig. 4). In the *internal control loop* pattern, the manager is a wrapper for the managed component and it is not adaptable. Instead, in the *external control loop* pattern, the manager is an adaptable component that is connected with the managed component. The distinction between external and internal control loops is also discussed in [Salehie and Tahvildari 2009], where it is stressed that internal control loops offer poor scalability and maintainability due to the intertwining of the application and the adaptation logic. Indeed this contradicts the separation-of-concerns principle that the authors (and many others) promote as key feature of self-adaptive systems. Like for MAPE-K, also for these control-loop centered

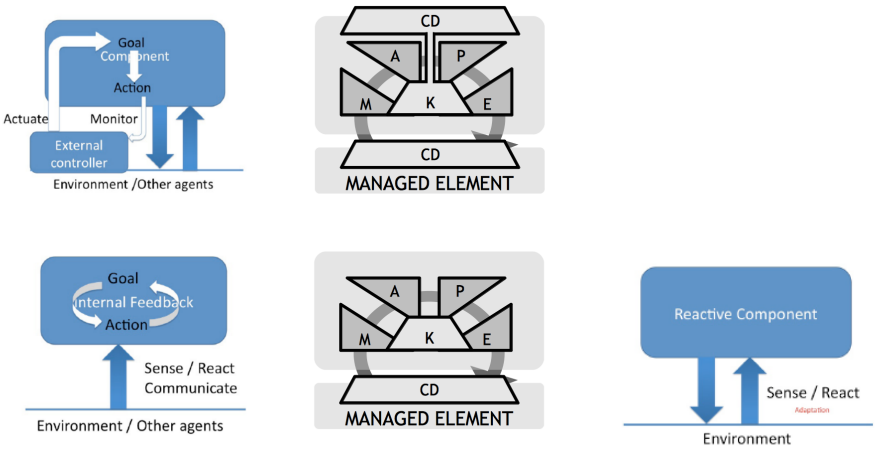

Fig. 4. External (top-left) and internal (bottom-left) control loop patterns and their presentation in terms of the MAPE-K model (center), and the reactive pattern (right).

approaches to adaptivity a precise identification of control data is only possible in concrete instances.

The taxonomy of [Cabri et al. 2011] includes a third pattern called *reactive pattern* that describes *reactive* components capable of modifying their behavior in reaction to an external event, without any control loop (or, equivalently, with a degenerate, "empty" control loop). In order to apply our definition of adaptation as *runtime modification of control data* to a reactive system of this kind, one could simply identify as control data those data that, when modified by sensing the environment, cause an adaptation of the system. This is a good example of the generality of our definition of adaptation, which is applicable also to such quite extreme case.

Another general reference model has been proposed in [Andersson et al. 2009b], where *computational reflection* is promoted as a necessary criterion for any self-adaptive software system. Reflection implies the presence, besides of base-level components and computations, of meta-level subsystems and meta-computations that act on a meta-model. Meta-computations can inspect and modify the meta-model that is causally connected to the base-level system, so that changes in either one are reflected in the other. The authors argue that most methodologies and frameworks proposed for the design and development of self-adaptive systems rely on some form of reflection, even if this is not always made explicit. Building on these considerations, they introduce the FOrmal Reference Model for Self-adaptation (FORMS) [Weyns et al. 2012], which provides basic modeling primitives and relationships among them, suitable for the design of self-adaptive systems (cf. Fig. 5). Such primitives allow one to make explicit the presence of reflective (meta-level) subsystems, computations and models.

The goals of [Andersson et al. 2009b] are not dissimilar from ours, as they try to capture the essence of self-adaptive systems, identifying it in computational reflection (one of the key features of self-adaptive systems according to [McKinley et al. 2004] as well). The FORMS modeling primitives can be instantiated and composed in a variety of ways. For example, [Weyns et al. 2012] provides one example that conforms to the MAPE-K reference model and another one that follows an application-specific design.

A precise identification of control data according to the criterion explained in Section 2 depends on the specific instance of the approach, and more precisely on the way modifications to the meta-level affect the base level, causing an adaptation. In instances

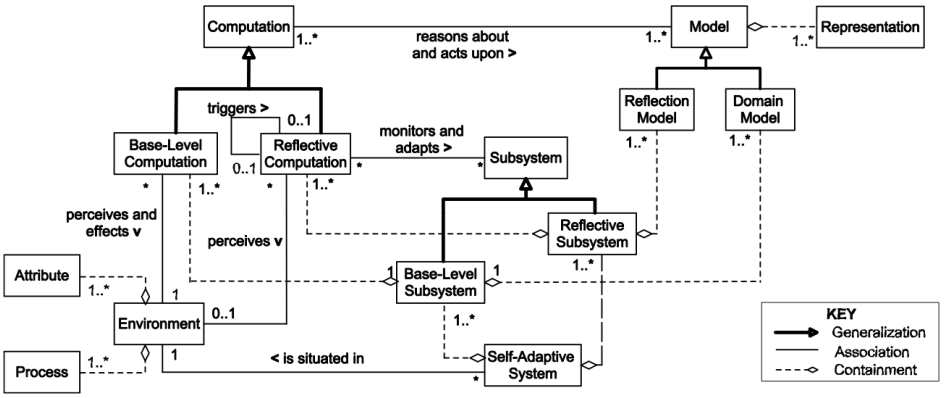

Fig. 5. The FORMS reference model.

featuring some kind of hot-linking from the meta- to the base-level component, the meta-level itself can be considered as control data. Otherwise, in general, control data will be identified at the boundary between the meta-level and the base-level components.

3.2. Reconfiguration-based Approaches to Adaptation

Several approaches to the design of (self-)adaptive systems look at a system as a network of components, suitably arranged in a logical or physical topology that constraints the interactions or communications among components. Adaptations in this context are typically realized via *reconfigurations*, which can range from the replacement of a single component to local or even global changes to the interaction topology. Usually such reconfigurations do not modify the functionalities of the individual components, but only the way they are connected and/or interact with each other (see the survey [Bradbury et al. 2004], summarized in Section 6, and [Kramer and Magee 2009]). Therefore the control data in these approaches can be identified with the interconnection topology itself, which depending on the approaches can be made of channels, connectors, gates, protocol stacks, links, and so on.

A first example is the approach presented in [Oreizy et al. 1999], where dynamic software architecture has a dominant role. The proposed methodology combines an Adaptation Management loop, which is essentially a distributed, agent-based MAPE-K control loop, with an Evolution Management loop. In the latter, an architectural model is maintained at runtime, that describes the running implementation and that plays the role of our control data. In fact the architectural model, made of components and connectors, can be modified by the control loop, by adding or removing components or connectors or by changing the topology. An Architecture Evolution Manger mediates the changes of the architectural model and maintains the consistency between the model and the running implementation.

The Ensemble system [van Renesse et al. 1998] is a network protocol architecture conceived with the aim of facilitating the development of adaptive distributed applications. The main idea is that each component of the application relies on a reconfigurable stack made of simple micro-protocol modules, which implement different component-to-component communication features. The module stack imposes a layered structure to the communication infrastructure which is used to guide its adaptation. For instance, adaptation can be triggered in a bottom-up way, when a layer n discovers some environmental changes that require an adaptation. Then the module at layer n may be

adapted and, if not possible, the adaptation request is propagated to the upper layer $n + 1$. Such structure is also exploited when a coordinated, distributed adaptation is needed, which is tackled by the *Protocol Switching Protocol*, one the key features of the approach. The protocol is initiated by a global coordinator that sends the notification of the need of adaptation to each component. Within each component the notification is propagated through the protocol stack, so that each layer applies the necessary actions. Adaptation can happen at different points. In particular it may affect the components participating to the distributed application (or to groups within it) or the communication infrastructure (i.e. the module stack). Hence, generally speaking, the set of components, their state and the module stack form the control data of the adaptive application.

The authors of [Karsai and Sztipanovits 1999] propose a model-based approach, showing how their approach to Model-Integrated Computing can be applied to adaptive systems. Adaptation is mainly reconfiguration followed by automatic deployment, triggered at runtime either by the user or by the system as reaction to certain events. In the proposed case study, a simple finite state automaton determines the transitions from one behaviour to another: in this case study the natural choice of control data consists of the states of the finite-state automaton.

The authors of [Bucchiarone et al. 2010] define a life-cycle for service-based applications where adaptation is a first-class concern. Such life-cycle continues during runtime, in order to cope with dynamic requirements and the corresponding adaptations. In addition to the life-cycle, the authors focus on the identification of a number of design principles and guidelines that are suitable for adaptable applications. Essentially, adaptation is understood as the modification of the workflow implementing a service-based application, from substituting individual services by equivalent ones, to recomposing a piece of the workflow to obtain an equivalent result. Therefore, roughly speaking, the current workflow is the control data of the service-based applications.

As a last example we consider the architectural approach of [Andrade and Fiadeiro 2002]. There, a system specification has a two-layered architecture to enforce a separation between computation and coordination. The first layer includes the basic computational components with well-identified interfaces, while the second one is made of connectors (called *coordination contracts*) that link the components appropriately in order to ensure the required system's functionalities. Adaptation in this context is obtained by reconfiguration, which can consist of removal, addition or replacement of both base components and connectors among them. The possible reconfigurations of a system are described declaratively with suitable rules, grouped in *coordination contexts*: such rules can be either invoked explicitly, or triggered automatically by the verification of certain conditions. In this approach, as adaptation is identified with reconfiguration, the control data consist of the whole two-layered architecture, excluding the internal state of the computational components.

4. COMPUTATIONAL MODELS FOR ADAPTATION

Computational reflection is widely accepted as one of the key instruments to model and build self-adaptive systems (cf. [McKinley et al. 2004; Dowling et al. 2000]). Indeed computational paradigms equipped with reflective, meta-level or higher-order features, allow one to represent programs as first-class citizens. In these cases adaptation emerges, according to our definitions, if the program in execution is represented in the control data of the system, and it is modified during execution causing changes of behaviour. Prominent examples of such formalisms are, e.g, rewrite theories with logical reflection like rewriting logic [Meseguer 1992] or process calculi with higher-order or meta-level aspects like HO π-calculus [Sangiorgi 1992]. Systems represented within

these paradigms can realize self-adaptation in a straightforward manner. Of course, computational reflection assumes different forms and, despite of being a very convenient mechanism, it is not strictly necessary: as we argued in Section 1 any programming language can be used to build a self-adaptive system.

We outline in this section some rules of thumb for the choice of control data within some well-known computational formalisms (deferring programming paradigms and languages to Section 5). In doing so, we restrict the attention to computational models that have been purposely introduced to represent adaptation and we argue how they can be used for modeling the behavior of self-adaptive systems. In addition, we survey a representative set of models that have been conceived with the specific purpose of modeling self-adaptive systems and supporting their formal analysis. We structure the presentation along three main strands: automata-like computational models (Section 4.1), declarative, rule-based computational models (Section 4.2), and computational models from the concurrency theory field (Section 4.3).

4.1. Automata-based Approaches to Adaptation

In many frameworks for the design of adaptive systems the base-level system has a fixed collection of possible behaviours (or behavioural models), and adaptation consists of passing from one behaviour to another. Some of the approaches discussed in this section achieve this by relying on a multi-layered structure reminiscent of hierarchical state machines and automata.

A first example of this tradition are the Adaptive Featured Transition Systems (A-FTS) of [Cordy et al. 2013], which were introduced for the purpose of model checking adaptive software (with a focus on software product lines). A-FTSs are a sort of transition systems where states are composed by the local state of the system, its configuration (set of *active features*) and the configuration of the environment. Transitions are decorated with executability conditions that regard the valid configurations. Adaptation corresponds to reconfigurations (changing the system's features). Hence, in terms of our white-box approach, reconfigurable system features play the role of control data. The authors introduce the notion of *resilience* as the ability of the system to satisfy properties despite of environmental changes (which essentially coincides with the notion of black-box adaptivity of [Hölzl and Wirsing 2011]). Properties are expressed in AdaCTL, a variant of the computation-tree temporal logic CTL.

Another example of layered computational structures are S[B] systems [Merelli et al. 2012], a model for adaptive systems based on 2-layered transitions systems. The base transition system B defines the ordinary (and adaptable) behavior of the system, while S is the adaptation manager, which imposes some regions (subsets of states) and transitions between them (adaptations). Further constraints are imposed by S via adaptation invariants. Adaptations are triggered to change *region* (in case of local deadlock). Such regions, hence, form the control data of the system according to our white-box approach. The paper also introduces formal notions of *weak* and *strong* adaptability, defined as the ability to conclude a triggered adaptation in some or all possible behaviors, respectively, and characterized by suitable CTL formulae.

Mode automata [Maraninchi and Rémond 1998] have been also advocated as a suitable model for adaptive systems. For example, the approach of [Zhao et al. 2011] represents adaptive systems with two layers: a *functional layer*, which implements the application logic and is represented by state machines called *adaptable automata*, and an *adaptation layer* that implements the adaptation logic and is represented with a mode automaton. Adaptation here is the change of mode, and these are the control data of this approach. The approach considers three kinds of specification properties: *local* (to

be satisfied by the functional behavior of one particular mode, not involving adaptation), *adaptation* (to be satisfied by adaptation phases, i.e. transitions between modes), and *global* (to be satisfied by all behaviors). An extension of linear-time temporal logic (LTL) called mLTL is used to express such properties.

Overlap adaptations [Biyani and Kulkarni 2008] arise in long-running open and dynamic distributed applications where components can be removed, added or replaced with a certain frequency. Under these premises, it is clear that the set of components of the application corresponds to its control data.

An overlap adaptation occurs when the execution of *old* components (i.e. components that need to be adapted) overlaps with the execution of *new* components (i.e. adapted components). This overlap introduces non-trivial issues but is required in order to adapt the whole application in a distributed manner without stopping it. The authors identify several kinds of overlap adaptations which vary in the kind of allowed interactions between old and new components.

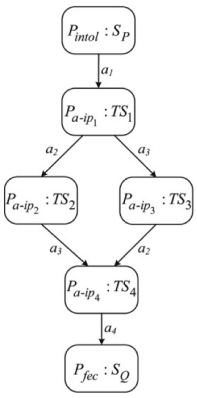

Fig. 6. An adaptation lattice.

The main concern of the approach is verifying the correctness of adaptations. For this purpose the approach relies on the concept of *transitional adaptation lattices*. Roughly, they are diamond-shaped graphs whose nodes represent automata and whose transitions correspond to atomic adaptation actions (cf. Fig. 6). Each automaton represents the behavior of the whole system in some state. The *top* automaton corresponds to the system before adaptation starts, while the *bottom* automaton corresponds to the system when adaptation ends. The diamond shape of the lattice implicitly imposes a confluent behavior of individual atomic adaptations.

Actually, the approach considers a finer granularity of components in terms of *fractions*, which are essentially the local instances of components in process locations. This fine-grained granularity introduces a combinatorial explosion in the size of the lattices which has a negative impact in the effort required in their correctness verification. To mitigate this the authors propose a framework based on particular architectures and coordination protocols, where some specialized modules can drive the adaptation phase through designated paths in the adaptation lattices. This implicitly introduces a higher-level adaptation since a system may vary the strategy of such modules according to various factors. In this case the control data of the system correspond to such strategies.

Another example of labelled transition system variant used for modeling self-adaptive systems are the *Synchronous Adaptive Systems* of MARS [Adler et al. 2007; Schaefer and Poetzsch-Heffter 2006], where systems are modeled as sets of modules, each having a set of configurations. At runtime only one configuration is active. Adaptation consists on changing the active configuration, selected according to the configuration conditions and the current environment. Control data in this approach are exactly those data that determine the active configuration.

While the "programs-of-programs" spirit can raise scalability and complexity issues, the layered structure of some of the above models can be exploited to study adaptive systems compositionally. The authors of [Zhang et al. 2009] propose a technique to verify properties of adaptive systems in a modular way. Adaptive programs are modeled with *n-plex adaptive programs* which are essentially sets of finite state machines, some of which representing *steady state programs* [Allen et al. 1998] and the rest representing adaptation transitions between those programs. The structure of an n-plex adaptive program makes explicit the separation of functional concerns (realized by steady state

programs) and adaptation concerns (realized by adaptation transitions), which is exploited to reason about such systems in a modular way. Clearly, the separation of concerns coincides with the spirit of CODA. In particular, control data here are the individual steady state programs.

This separation of concerns has its counterpart in the property specification language used, namely *Adapt-operator extended LTL* (A-LTL) proposed in [Zhang and Cheng 2006b]. A-LTL extends LTL with an adapt operator that does not provide more expressive power but allows to express properties of adaptive systems in a significantly more concise manner. With respect to similar approaches, the modular verification phase exploits the separation of concerns and the assume/guarantee paradigm in order to avoid the state explosion problem, thus providing a more scalable solution. For instance, this allows the authors to tackle *transitional* properties of adaptation (e.g. graceful adaptation, hot-swapping adaptation, restriction of adaptations to quiescent states, etc.) in an efficient manner.

Structuring the behavior of adaptive system is a major concern in [Iftikhar and Weyns 2012]. The authors identify four main modes of operation (called *state space zones*) in an adaptive system: the *normal* behavior zone (the system operates as expected), the *undesired* behavior zone (the system has violated some constraint and needs to be adapted), the *invalid* behavior zone (the system has violated some constraint and cannot be adapted), and the *adaptation* behavior zone (the system is adapting to re-enter the normal behavior zone). Their work is motivated by the necessity of shifting the focus to behavioral aspects of adaptation, as evidenced in previous experiences of the authors [Weyns et al. 2012] that were mainly concerned with architectural aspects. In this approach, hence, the control data are those used to characterize the state space zones. The authors use their approach to model and analyze the case study of a decentralized adaptive traffic control system using timed automata and TCTL, a timed extension of CTL. The authors distinguish two different adaptation capabilities (from the black-box perspective): *flexibility* (ability to adapt to changing environments, e.g. to improve performance) and *robustness* (ability to recover from failures).

Some of the above approaches rely on logical reasoning mechanisms to prove properties of adaptation. To this end, base steady programs are annotated with the properties they ensure (cf. the above discussed adaptation lattices [Biyani and Kulkarni 2008]). This idea of specification-carrying programs is investigated in [Pavlovic 2000]. The author identifies suitable semantical domains aimed at capturing the essence of adaptation. The behaviour of a system is formalized in terms of a category of specification-carrying programs (also called *contracts*), i.e. triples made of a program, a specification and a satisfaction relation among them; arrows between contracts are refinement relations. Contracts are equipped with a functorial semantics, and their adaptive version is obtained by indexing the semantics with respect to a set of *stages of adaptation*, yielding a coalgebraic presentation potentially useful for further generalizations. An adaptation is a transformation of a specification-carrying-program into another one, satisfying some properties. Therefore, the control data includes the entire program being executed.

Different in spirit is our proposal in [Bruni et al. 2013] where we studied the consequences of making a particular choice of control data in automata-like models (and, in particular, in Interface Automata [de Alfaro and Henzinger 2001]). For this purpose we introduced the concept of Adaptable Transition Systems and its instantiation to Adaptable Interface Automata (AIA), an essential model of adaptive systems inspired by the white-box approach to adaptation discussed here, and based on a foundational model of component-based systems. The key feature of AIAs are control propositions,

the formal counterpart of control data. The choice of control propositions is arbitrary, but it imposes a clear separation between ordinary, functional behaviors and adaptive ones. We discuss how AIAs can be exploited in the specification and analysis of adaptive systems, focusing on various notions proposed in the literature, like adaptability, feedback loops, and control synthesis.

4.2. Rule-based Models for Adaptation

Rule-based programming is an excellent example of a successful and widely adopted declarative paradigm, thanks to the solid foundations offered by rule-based theoretical frameworks like term and graph rewriting. As many other programming paradigms, several rule-based approaches have been tailored or directly applied to adaptive systems (e.g. graph transformation [Ehrig et al. 2010]). Typical solutions include dividing the set of rules into those that correspond to ordinary computations and those that implement adaptation mechanisms, or introducing context-dependent conditions in the rule applications (which essentially corresponds to the use of standard configuration variables). The control data are identified by the above mentioned separation of rules in the first case, and they correspond to the context-dependent conditions in the latter.

The situation is different when we consider rule-based approaches which enjoy higher-order or reflection mechanisms. A good example is *logical reflection*, a key feature of frameworks like rewriting logic [Meseguer 1992]. At the ground level, a rewrite theory $\mathcal{R}$ (e.g. a software module) lets us infer a computation step $\mathcal{R} \vdash t \to t'$ from a term (e.g. a program state) t into t'. A universal theory $\mathcal{U}$ lets us infer the computation at the "meta-level", where theories and terms are meta-represented as terms: the above computation step can be expressed in $\mathcal{U}$ as $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}}, \overline{t'})$; moreover, the rewrite theory $\mathcal{R}$ can be also rewritten by meta-level rewrite rules, like in $\mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \to (\overline{\mathcal{R}'}, \overline{t'})$. Since $\mathcal{U}$ itself is a rewrite theory, the reflection mechanism can be iterated yielding what is called the *tower of reflection*, where not only terms $\overline{t}$, but also rewrite rules of the lower level can be accessed and modified at runtime. This mechanism is efficiently supported by Maude [Clavel et al. 2007] and has given rise to many interesting meta-programming applications like analysis and transformation tools.

In particular, the reflection mechanism of rewriting logic has been exploited in [Meseguer and Talcott 2002] to formalize a model for distributed object reflection, suitable for the specification of adaptive systems. Such model, suggestively called Reflective Russian Dolls (RRD), has a structure of layered configurations of objects, where each layer can control the execution of objects in the lower layer by accessing and executing the rules in their theories, possibly after modifying them, e.g. by injecting some specific adaptation logic in the wrapped components (cf. Fig. 7). Even at this informal level, it is pretty clear that the RRD model falls within our conceptual framework by identifying as control data for each layer the rules of its theory that are possibly modified by the upper layer. Note that, while the tower of reflection relies on a white-box architecture, the Russian Dolls approach can deal equally well with black-box components, because wrapped configurations can be managed by message passing. The RRD model has been further exploited, among others, for modeling policy-based coordination [Talcott 2006], for the design of PAGODA, a modular architecture for specifying autonomous systems [Talcott 2007], in the composite actors used in [Eckhardt et al. 2013], and, by ourselves, in the design and analysis of self-assembly strategies for robot swarms [Bruni et al. 2012b].

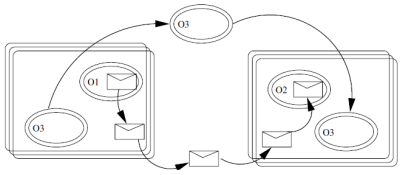

Fig. 7. RRDs.

The RRD approach is also very suitable to model *self-awareness* in software systems, intended as the means by which a software system is *"aware of its self states and behaviors"* [Hinchey and Sterritt 2006]. Indeed, the *self states* can be modeled as the meta-representation of the current state of the objects, while the *self behaviors* can be modeled as the meta-representation of the objects rules.

4.3. Concurrency Models for Adaptation

Languages and models conceived in the area of concurrency theory are also good candidates for the specification and analysis of self-adaptive systems. We inspect some paradigmatic formalisms to see how the conceptual framework can help us in the identification of the adaptation logic within each model.

Petri nets are undoubtedly the most popular model of concurrency, based on a set of repositories, called places, and a set of activities, called transitions. The state of a Petri net is called a marking, that is a distribution of resources, called tokens, among the places of the net. A transition is an atomic action that consumes several tokens and produces fresh ones, possibly involving several repositories at once. In *coloured* Petri nets, the tokens can represent structured data and transitions can manipulate them.

The approach proposed in [Zhang and Cheng 2006a] emphasizes the use of Petri nets to validate the development of adaptive systems.

Specifically, it represents the local behavioural models with coloured Petri nets, and the adaptation change from one local model to another with an additional Petri net transition labeled `adapt` (cf. Fig. 8). Such `adapt` transitions describe how to transform a state (a set of tokens) in the source Petri net into a state in the target model, thus providing a clean solution to the *state transfer problem* (i.e. the problem to transfer the state of the system before and after the adaptation in a consistent way) common to these approaches. In this context, a good choice of control data would be the Petri net that describes the current base-level computation, which is replaced during an adaptation by another local model.

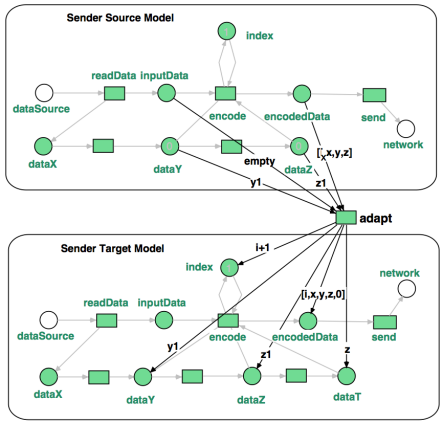

Fig. 8. Petri net model of an adaptive system.

Petri nets are also exploited in [Popescu et al. 2012] to formalize multi-layer adaptation in large scale applications that can span over heterogeneous organizations, technologies and devices. Here the multi-layered architecture is motivated by the presence of different languages and technologies addressing their own concerns and views within the same application in a coherent manner and multi-layered adaptation must ensure that coherence between views is always maintained. For example, a three-layers architecture is typical of service-based applications: one layer for service specification (e.g., WSDL); one layer for behavior description (e.g., BPEL); and the organizational view that specifies the stakeholders involved in the business process.

Multi-layer adaptation is triggered by *adaptation events* that are raised by human stakeholders or by layer-specific monitors that discover, e.g., message-ordering mismatches (at the behavior level), or invocation mismatches (at the service layer). Application mismatches are organized along tree-based taxonomies that are put in correspondence with suitable adaptation templates. The main idea is that adaptation techniques that can tackle one application mismatch m can also be used to adapt mismatches that

are "below" m in the taxonomy. Cross-layer adaptation is achieved by linking templates either at different application layers: templates may trigger the executions of other templates both through direct invocation or by raising other adaptation events,

Adaptation templates, the taxonomy navigation and the template-selection environment are modeled as Petri nets (they support the search of the templates starting from the more specific to the more general, w.r.t. the raised adaptation event). As the emphasis is the specification of a generic adaptation model for pervasive applications, the Petri net model abstracts away from the execution of multi-layered applications and thus the identification of control data is only possible for concrete instances of this model.

Classical process algebras (CCS, CSP, ACP) are certainly tailored to the modeling of reactive systems and therefore their processes easily fall under the hat of the reactive pattern of adaptation. Instead, characterizing the control data and the adaptation logic is more difficult in this setting. The π-calculus [Milner 1999], the join calculus [Fournet and Gonthier 2002] and other nominal calculi, including higher-order versions (e.g. the HO π-calculus [Sangiorgi 1992]) can send and receive channels names, realizing some sort of reflexivity at the level of interaction: they have the ability to communicate transmission media.

An example of the use of the π-calculus for modeling autonomic computing systems can be found in [Wang et al. 2009]. There, adaptive systems are organized in two-levels: the local level and the global one. The local level is formed by autonomic elements structured in the MAPE-K spirit as a managed element and an autonomic manager, all defined by π-calculus processes that communicate over designated channels. In particular, the effector process enacts adaptation requests by sending messages to its managed element over the effector channel, which can be understood as the control data of the local adaptive behavior. At the global level a centralized autonomic manager monitors and controls the local distributed autonomic managers. Again, adaptation is realized by sending messages through suitable effector channels.

Similar approaches have been explored within process calculi that feature primitives that seem adequate to model autonomic systems, including explicit locality aspects, asynchronous communication and code mobility (e.g. based on tuple-spaces). A paradigmatic example is the KLAIM process algebra [De Nicola et al. 1998], which has been studied as a convenient mechanism for modeling self-adaptive systems in [Gjondrekaj et al. 2012]. The authors describe how to adopt in KLAIM three paradigms for adaptation: two that focus on the language-level, namely, context-oriented programming and aspect-oriented programming (that are discussed in Sections 5.1 and 5.2, respectively), and one that focuses on the architectural-level (i.e. MAPE-K).

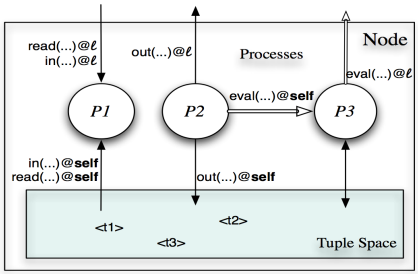

Fig. 9. A KLAIM node.

The main idea in all cases is to rely on the use of *process tuples*, that is, tuples (the equivalent of messages in the tuple-space paradigm) that denote entire processes. These process tuples can be sent by manager components (locations in KLAIM) to managed components, which can then install them via the *eval* primitive of KLAIM (cf. Fig. 9). In other words, adaptation is achieved by means of code mobility and code injection. The control data in this case amounts to the set of active processes in each location. Indeed, adaptation in their view is the act of installing a new process.

Stemming from this approach, the Service Component Ensemble Language (SCEL) has been proposed in [De Nicola et al. 2013] which realizes adaptation by combining different paradigms, namely policy-based programming (discussed in Section 5.3), tuple-space communication, and knowledge-based reasoning. In this case control data is spread among the policy rules, the process tuples and the knowledge facts and clauses.

In [Martín et al. 2012] the authors present a lightweight approach to service adaptation based on process algebraic techniques. Likewise [Bracciali et al. 2005], adaptation is achieved by the design-time synthesis of service adaptors that act as mediators for the communication between two services and allow to overcome signature and behaviour mismatches between their contracts. Differently from [Bracciali et al. 2005], an adaptor process is deployed that is itself adaptive, in the sense that its behaviour is initially distilled on the basis of adaptation contracts and then the adaptor is progressively refined at run-time exploiting the collected information about interaction failures. This is useful when service behavior may evolve at runtime due to changes of the environmental conditions in ways not foreseeable in the contract, e.g. depending on the current load of its server. The approach is lightweight because it introduces low overhead. Learning adaptors have been implemented and included in the Integrated Toolbox for Automatic Composition and Adaptation (ITACA) [Cámara et al. 2009]. The control data of the approach are the adaptors themselves.

We conclude this section by mentioning the approach in [Bravetti et al. 2012], where the concept of *adaptable process* has been put forward to model dynamic process evolution patterns in process algebras. Adaptable processes are assigned a location and can be updated at runtime by executing an update prefix related to that location. Roughly, if P is an adaptable process running at location a, written $a[P]$, and U is a process context, called *update pattern*, then the execution of the update prefix $\tilde{a}\{U\}$ stops the execution of P within a (i.e., $a[P]$ is removed) and replaces it with $U(P)$. Note that location a is not necessarily preserved by the update, providing flexibility on the allowed update capabilities. For example, the prefix $\tilde{a}\{nil\}$ would just remove $a[P]$; the prefix $\tilde{a}\{a[Q]\}$ would replace $a[P]$ by $a[Q]$; the prefix $\tilde{a}\{b[\cdot]\}$ would move P from location a to the location b; and the prefix $\tilde{a}\{a[\cdot|\cdot]\}$ would spawn an extra copy of P within a. The authors exploit the formal model to study undecidability issues of two verification problems, called *bounded* and *eventual adaptation*, i.e., that there is a bound to the number of erroneous states that can be traversed and that whenever a state with errors is entered, then a state without errors will be eventually reached, respectively. The control data of [Bravetti et al. 2012] are the adaptable processes of the form $a[P]$.

5. PROGRAMMING PARADIGMS FOR ADAPTATION

As observed in the previous sections, the nature of control data can vary considerably depending both on the degree of adaptivity of the system and on the nature of the computational formalisms used to implement it. Examples of control data include configuration variables, rules and plans (in rule-based programming), code variations (in context-oriented programming), interactions (in connector-centered approaches), policies (in policy-driven languages), advices (in aspect-oriented languages), monads and effects (in functional languages), and even entire programs (in models of computation exhibiting higher-order or reflective features). Indeed, many programming languages that consider such forms of control data as first-class citizens have been promoted as suitable for programming adaptive systems (see the overviews of [Ghezzi et al. 2011; Salvaneschi et al. 2013]). Just restricting to Java some examples of technologies supporting adaptation include Jolie [Montesi et al. 2007], ContextJ [Appeltauer et al. 2011], JavAdaptor [Pukall et al. 2013] and Chameleon [Autili et al. 2010]. We survey in this section a representative set of such programming paradigms and explain their notion

of adaptation in terms of CODA. In particular, we organize the discussed approaches in three paradigms: context-oriented programming (Section 5.1), aspect-oriented programming (Section 5.2), and policy-oriented programming (Section 5.3).

5.1. Context-Oriented Programming for Adaptation

Context-oriented programing [Hirschfeld et al. 2008] has been designed as a convenient paradigm for programming autonomic systems [Salvaneschi et al. 2011]. The main idea of this paradigm is to rely on a pool of *code variations* choosen according to the program's *context*, i.e. the runtime environment under which the program is running. Under this paradigm the natural choice of control data is the current set of active code variations.

Many languages have been extended to adopt the context-oriented paradigm. We mention among others Lisp, Python, Ruby, Smalltalk, Scheme, Java, and Erlang. The notion of context varies from approach to approach and in general it might refer to any computationally accessible information. Without giving any concrete reference, a typical example is the environmental data collected from sensors. In many cases the universe of all possible contexts is discretised in order to have a manageable, abstract set of fixed contexts. This is achieved, for instance by means of functions mapping the environmental data into the set of fixed contexts. Code fragments like methods or functions can then be specialized for each possible context. Such chunks of behaviours associated with contexts are called *variations*.

The context-oriented paradigm can be used to program autonomic systems by activating or deactivating variations in reaction to context changes. The key mechanism exploited here is the dynamic dispatching of variations. When a piece of code is being executed, a sort of dispatcher examines the current context of the execution in order to decide which variation to invoke. Contexts thus act as some sort of possibly nested scopes. Indeed, very often a stack is used to store the currently active contexts, and a variation can propagate the invocation to the variation of the enclosing context.

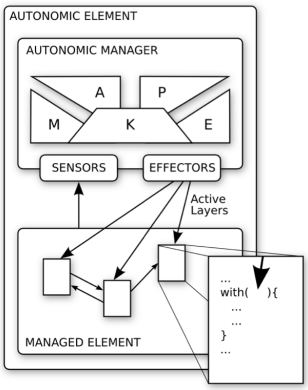

Fig. 10. MAPE-K architecture in context-oriented programming.

The key idea to achieve adaptation along the lines of the MAPE-K framework is for the manager to control the context stack (for example, to modify it in correspondence with environmental changes) and for the managed component to access it in a read-only manner. Those points of the code in which the managed component queries the current context stack are called *activation hooks* (*adaptation hooks* in [Lanese et al. 2010] and in [Gjondrekaj et al. 2012], as we shall see in Sections 5.2 and 5.3, respectively).

Given the above informal description, context-oriented programming falls into CODA by considering the context stack as control data. With this view, the only difference between the approach proposed in [Salvaneschi et al. 2011] (cf. Fig. 10) and our ideas is that the former suggests the context stack to reside within the manager (this is not clear in the figure, but we refer to the detailed example in the cited paper), while we advocate the control stack to reside in the interface of the managed component, so to be able to identify the managed component as an adaptable component.

5.2. Aspect-Oriented Programming for Adaptation

Aspect-oriented programming [Kiczales et al. 1997] and, in particular, dynamic aspect-oriented programming [Popovici et al. 2003] have been advocated as a convenient

mechanism for the development of self-adaptive software by many authors since the original proposal of [Greenwood and Blair. 2004].

The main idea is that the separation-of-concerns philosophy of aspects facilitates the addition of autonomic computing capabilities to software systems. Indeed, while early works [Greenwood and Blair. 2004] put the stress on monitoring as an aspect, subsequent works have generalized this idea to other capabilities. Adaptation, for instance, can be realized through aspect weaving, i.e. the activation and deactivation of advices (the code to be executed at join points), possibly enacted by an autonomic manager. Advices, hence, can be understood as the control data of the aspect-based adaptation paradigm. Dynamic aspect oriented programming languages, which are equipped with dynamic aspect weaving mechanisms, thus facilitate the realization of dynamic adaptation.

5.3. Policy-Oriented Programming for Adaptation

As we have seen in Section 4.2, rule-based approaches have been advocated as a convenient mechanism for realizing self-adaptation. Another example of this tradition are policies. Generally speaking, policies are in fact rules that determine the behavior of an entity under specific conditions. Policies have been seen as mechanisms enjoying the flexibility required by self-* systems, and tackling the problem at the right (high-) level of abstraction. Quite naturally, adaptation can be realized by changing policies according to the program's current status. The natural choice of control data is then the current set of active policies.

A prominent example is the Policy-based Self-Adaptive Model (PobSAM) [Khakpour et al. 2012], a formal framework for modeling and analyzing self-adaptive systems which relies on policies as a high-level mechanism to realize adaptive behaviors. Building upon the authors experience in the development of the PAGODA framework [Talcott 2007] (cf. Section 4.2), PobSAM combines the actor model of coordination [Agha 1986] with process algebra machinery and shares the white-box spirit of separating application and adaptation concerns. Indeed, the overall architecture of the system is composed by *managed actors*, which implement the functional behavior of the system, and *autonomic manager (meta-)actors*, which control the behavior of managed actors by enforcing policies. In this manner, the adaptation logic is encoded in policies whose responsibility relies on well-identified system components (i.e. the managers). In particular, the configuration of managers is determined by their sets of policies which can vary dynamically. The currently active set of policies represents the control data in this approach. Adaptation is indeed the switch between active policies. Policies are rules that determine under which condition a specified subject must or must not do a certain action. PobSAM distinguishes between *governing* policies, which control the managed actors in their *stable* (cf. steady, normal) state and *adaptation* policies, which drive the actors in the transient states (cf. adaptation phases).

The authors of [Lanese et al. 2010] propose a framework for dynamic adaptation based on the combination of *adaptation hooks*, which specify *where* to apply adaptation, and policies called *adaptation rules*, which specify *when* and *how* to apply it. In their approach an adaptable application is an application that exposes part of its states and the set of activities that it performs in a suitable interface called *application interface*. Adaptation is enacted by suitable managers that exploit the adaptation rules in order to introduce changes in the application through its interface. In particular, the rules define adaptations that may change the activities by instantiating new code or changing their configuration parameters and may also change part of the application's state. Hence, in this approach both the set of activities and the exposed application state are to be considered as control data in the basic adaptation layer. On top of this basic layer,

dynamic adaptation can occur, which consists on modifying the adaptation rules at runtime. This makes adaptation managers adaptable as well. At this layer, hence, the control data are precisely the adaptation rules, which determine the behavior of the adaptation managers.

The approach is instantiated in the Java Orchestration Language Interpreter Engine (Jolie) [Montesi et al. 2007] a framework for rapid prototyping of service oriented applications. The approach is, however, language agnostic. As a matter of fact, the authors identify the basic ingredients needed to implement their approach in other settings and a generic architecture to structure the framework. The former consists of mechanisms needed to implement the adaptation interface and its manipulation based on code mobility. At the architectural level applications are structured as *clients* which rely on an *activity manager* to run their activities. Adaptation is governed by *adaptation servers*, which are coordinated globally by an *adaptation manager service*.

6. RELATED WORK

We have already discussed some of our sources of inspiration in the previous sections and spelled out how their underlying notion of adaptation can be recast in terms of our approach. This section overviews and discusses three kinds of related works. Section 6.1 is devoted to works that propose a definition of adaptation. Section 6.2 overviews and discusses research works that provide a classification of approaches and techniques, guided by a set of dimensions or facets relevant to adaptive systems. Section 6.3 provides some further interesting pointers to the literature. We are aware that the references considered here represent only a fragment of the vast literature on adaptive systems which, for obvious reasons, we cannot discuss here in a comprehensive manner. We refer the interested reader to the bibliography of the surveys discussed in this section for completing the picture.

6.1. On the Essence of Adaptation

In this section we focus on approaches whose aim is similar to ours, namely to provide conceptual notions of adaptation. Several proposals follow a black-box perspective that, as discussed in the Introduction, focuses on the external observation of self-adaptive systems, i.e. at measuring or expressing requirements on how a software system changes its ability to reach a goal under specific context variations.

An interesting contribution of this kind is [Lints 2010], which analyses the notion of adaptation in a very general sense and identifies the main concepts around adaptation drawn from several different disciplines, including evolution theory, biology, psychology, business, control theory and cybernetics. Furthermore he provides general guidelines on the essential features of adaptive systems in order to support their design and understanding.

The author claims that *"in general, adaptation is a process about changing something, so that it would be more suitable or fit for some purpose that it would have not been otherwise"*. The author uses the term *adaptability* to denote the capacity of enacting adaptation, and *adaptivity* for the degree or extent to which adaptation is enacted. This leads to the identification of four main issues that typically play a role in approaches to adaptation: context, goals, time-frames, and granularity that are discussed in Section 6.2. The author concludes his discussion suggesting that *"due to the relativity of adaptation it does not really matter whether a system is adaptive or not (they all are, in some way or another), but with respect to what it is adaptive"*.

A formal black-box definition is proposed in [Broy et al. 2009]. If a system reacts differently to the same input stream provided by the user at different times, then

the system is considered to be adaptive because ordinary systems should exhibit a deterministic behavior. Thus, a non-deterministic reaction is interpreted as an evidence of the fact that the system adapted its behaviour after an interaction with the environment. Despite its appeal and crispness, we believe that this and similar definitions of adaptation are based on too strong assumptions, restricting considerably its range of applicability. For example, a system where a change of behaviour is triggered by an interaction with the user would not be classified as adaptive.

As we argued in the Introduction, black-box approaches are interesting and useful for evaluating the system robustness under some conditions. However, they are of little use for design purposes where modularization and reuse are critical aspects. Therefore, we believe that a formal definition of adaptation should not be based on the observable behaviour of systems only, as it happens in the black-box approaches. At the same time, we do believe that research efforts are needed to conciliate black-box and white-box perspectives. Ideally, the internal mechanisms and external manifestations of adaptive behavior should be coherent, so that, for instance, a black-box analysis can validate that the degree of adaptability is strongly dependent on the adaptation mechanisms.

A different perspective on adaptation, inspired by the seminal work of IBM on autonomic computing, has been adopted by many authors, e.g. [Salehie and Tahvildari 2009]). The starting point is the observation that modern software can be seen as an open loop. Indeed, a software system is inevitably subject to continuous modifications, reparations and maintenance operations which require human intervention. Self-adaptation is seen as the solution to such openness by closing the loop with feedback from the software itself and its context of operation. In this view self-adaptation is seen as a complex feature built upon self-awareness and other self-* mechanisms. Control loops are seen as a fundamental process to achieve adaptive behaviors.

The kind of adaptation discussed so far is concerned essentially with individual components. However it may also happen that a complex system made of non-adaptive components exhibits a collective behavior which is considered to be adaptive (see e.g. the discussion in [Lints 2010]). Such *emergent adaptation*, typical of massively parallel and distributed systems such as *swarms* and *ensembles*, is the result of the interaction among components. Very often, emergent adaptation relies on decentralized coordination mechanisms (e.g. based on the spatial computing paradigm [Viroli et al. 2011; Beal et al. 2012]). Interesting in this regard can be to shift the focus to *Singerian* forms of adaptation [Sagasti 1970; Bouchachia and Nedjah 2012], where the subject of adaptation is the environment, as opposed to the *Darwinian* adaptation we have focused on, where the system is the subject of the adaptation.

A conceptual framework for emergent adaptation would require to shift from a *local* notion of control data to a *global* one, where the control data of the individual components of the system are treated as a whole, possibly requiring some mechanism to amalgamate them for the manager and to project them backwards to the components.

6.2. The Facets of Adaptation

The literature on adaptive systems contains several interesting surveys and taxonomies based on the identification of what the authors consider to be the main facets of adaptation. The concept of control data provides one such facet that has been used in this paper to classify many proposals as discussed in Sections 3–5 and summarized in Fig. 1. In this section we relate control data with other facets proposed in the literature. In most cases these are orthogonal and provide therefore complementary classification criteria. In a few cases they are closely related with control data, thus providing a more concrete perspective on the corresponding approaches.

The survey on self-adaptive software of [Salehie and Tahvildari 2009] is one of the most comprehensive studies on the topic, including also approaches to adaptation from the fields of artificial intelligence, control theory and engineering, and decision theory. It presents a taxonomy of adaptation concerns, surveys a wide set of representative approaches from many different areas, and identifies some key research challenges. The discussion is driven by the so-called six honest men issues in adaptation: (1) *Why* is adaptation required? Is the purpose of adaptation to meet some robustness criteria, to improve the system's performance or to satisfy some other goal? (2) *When* should adaptation be enacted? Should adaptation be applied reactively or proactively? (3) *Where* is the need to do an adaptation manifested? That is, which artifacts (sensors, variables, etc.) indicate that it is necessary to perform an adaptation? (4) *What* parts of the system should be adapted? That is, which artifacts (variables, components, connectors, interfaces, etc.) have to be modified in order to answer to the adaptation needs? (5) *Who* should enact the adaptation? Which entity (e.g. human controller, autonomic manager) is in charge of each adaptation? (6) *How* should adaptation be applied? That is, which is the plan that establishes the order in which to apply the necessary adaptation actions?

Our conceptual framework fits well with this approach and is mainly devoted to the identification of the *what*, which then facilitates finding the right *who*, *why*, *when*, *where* and *how* of a system's adaptation mechanism. In fact, in our view the *what* corresponds precisely to the control data. Interestingly, the taxonomy distinguishes between *weak* adaptation (e.g. modifying parameters) and *strong* adaptation (e.g. replacing entire components): the granularity of control data obviously provides a finer spectrum between these two extremes.

The authors of [McKinley et al. 2004] identify and promote three key technologies that enable the development of adaptive systems and that are nowadays widely accepted: component-based design, separation of concerns, and computational reflection. We remark that our aim is more devoted to providing a common understanding of adaptation rather than promoting particular mechanisms.

They argue that there are two main approaches to adaptation: *parameter* adaptation and *compositional* adaptation. As already discussed for the distinction between weak and strong adaptation, also parameter and compositional adaptations can be casted to different choices of control data. Indeed, in parameter adaptation control data can be identified in those program variables that affect the system behavior, and adaptation coincides with the modification of those variables. Instead, in compositional adaptation control data can be identified in the system's architecture, i.e. in the system components and interconnection, and adaptation coincides with architectural reconfiguration, from replacing entire components to modifying only parts of them. The authors pay a special attention to compositional adaptation and propose a taxonomy that focuses on three main questions: the *when*, *how*, and *where* to compose.

While our aim is centered around the conceptual forms of control data, the authors focus on concrete technological mechanisms and do not consider foundational models such as those discussed in Section 4. In particular, they enumerate mechanisms such as wrappers, proxies, virtual component patterns, the meta-object protocol, aspect weaving (cf. Section 5.2), and middlewares.

The authors of FORMS (cf. the discussion on [Andersson et al. 2009b; Weyns et al. 2012] in Section 3) provide in [Andersson et al. 2009a] a classification of modeling facets for self-adaptive systems. The authors have focused on the implicit underlying conceptual models rather than on the concrete technologies used to realize them. As a result they identify four main groups of facets: those regarding the *goals* or objectives of adaptation, the *changes* that trigger adaptation, the *mechanisms* that realize the

adaptation, and the *effects* of adaptation. The proposed classes for each facet seem orthogonal with respect to the choice of control data.

The authors exploit such classes to identify the research challenges of adaptation. They stress, among others, the need of mechanisms to conciliate conflicting goals in open systems where participants may be in competition; the need of lightweight monitoring and adaptation techniques to mitigate their overhead; the need of decentralized mechanisms for coordinating adaptation in distributed systems; the need of responsive mechanisms in adaptive real-time systems; and the need of verification, validation, and prediction mechanisms to ensure that self-adaptive systems behave correctly and predictably.

The survey [Bradbury et al. 2004] provides an overview of those approaches that support self-adaptation based on architectural reconfiguration. The authors consider that an architecture is *self-managed* if it can perform architectural changes at runtime by initiating, selecting, and assessing them by itself, without the assistance of an external entity. Contrary to other surveys on architectural reconfiguration (e.g. [Clements 1996; Mikic-Rakic and Medvidovic 2006]) the authors focus on formal models such as graphs, process algebras and logic.

The considered approaches are evaluated in terms of their support for basic reconfigurations such as component or connector addition/removal and composite reconfiguration operations such as sequentialization, iteration and choices. With respect to our proposal, they clearly identify the software achitectures themselves as control data (cf. also the discussion in Section 3.2).

In [Lints 2010] four main facets of adaptation are proposed that arise from different disciplines. They are conceded with the context, goals, time-frames, and granularity of the system.

In control theory adaptation is a mechanism to deal with contextual perturbations. In other fields (e.g. robotics and psychology) the behavior of an entity is considered to be adaptive depending on the context in which such behavior is enacted or even in how regular such context is (cf. the example of this black-box perspective in the field of robotics [Harvey et al. 2005] in Section 2). In Computer Science also white-box approaches have been proposed that are centered around the notion of context (cf. the discussion on the context-oriented paradigm in Section 5.1.

Adaptation as an act aimed at fulfilling explicit goals is something common, but not mandatory. There are indeed examples of adaptive systems that are purely reactive and do not have an explicit goal to be pursued. This is often the case in control systems, where *robustness*, i.e. the ability to keep the system in some state in spite of external perturbations, is sometimes considered as an adaptation mechanism. Lofti Zadeh [Zadeh 1963], for instance, proposes to consider a system as adaptive with respect to operating conditions and a class of performance values if its performance in those conditions stays within that class.

Adaptation may regard either short- or long-term time frames. Long-term adaptation, sometimes called *evolution*, does very often involve evolutionary and learning processes. Evolutionary approaches are original from biology, where adaptation has been often seen as a selective process, which involves the notions of feedback loops and fitness functions, fundamental as well in many other fields, control theory among others.

Last but not least, related to the time-framing is the notion of granularity. Adaptation can be seen as a feature of individuals (e.g. survival in biological systems) or collectives (e.g. continuation of the species). As a matter of fact a system, even a software system, may be considered as adaptive even if made of components that are not considered to be adaptive. This is sometimes called *emergent* adaptation. We note that the classification based on control data is best suited for adaptation of individual components.

6.3. Other works

Other interesting overviews on adaptation include the research roadmaps on software engineering for self-adaptive systems described in [Cheng et al. 2009; de Lemos et al. 2011]; the proposal of [Raibulet 2008] to identify the facets of adaptation and the corresponding research challenges; the survey on context-aware service engineering of [Kapitsaki et al. 2009] which focuses on mechanisms for realizing context-awareness in adaptive services; the survey on autonomic computing of [Huebscher and McCann 2008], which provides a comprehensive overview of the past, present and future of autonomic computing research; the work of [Dobson et al. 2006] that focuses on communication mechanisms in autonomic computing; the discussion of [Mühl et al. 2007] which introduces a classification of such systems building upon Zadeh's definition of adaptive systems; the work reported in [Fritsch et al. 2008], which describes a classification of automotive software with respect to their adaptation requirements and defines a taxonomy of adaptation dimensions in automotive software; and the research roadmap and vision of the Descartes research group described in [Kounev 2011] that contains interesting ideas regarding the challenge for engineering the next generation of self-* systems and services based on extending he MAPE-K control loop to support online QoS prediction.

7. CONCLUSION

We have presented CODA, a white-box conceptual framework for adaptation that promotes a neat separation of the adaptation logic from the application logic through a clear identification of control data and their role in the former logic. To validate CODA we have described a representative set of approaches to (self-) adaptation ranging from architectural solutions (Section 3), to computational models (Section 4), and to programming languages and paradigms (Section 5). For each of them we have highlighted the main distinguishing features and we have discussed the way they fit in CODA. As a byproduct, our work provides an original perspective from which to survey Computer Science approaches to adaptive systems. As a matter of fact, we have also discussed (Section 6) other surveys and taxonomies conceived with the same aim as our work: to establish a common ground for fruitful research debates by clarifying and identifying the key features of adaptive systems.

The discussion of this paper has also helped us to identify many different forms of control data that can be found in the literature. Our position is that *the* best form of control data does not exist. Every form of control data can be adequate. However, we strongly believe that the choice of control data should adhere to the following three principles (cf. [McKinley et al. 2004]): separation of concerns, component-based design and computational reflection.

Regarding the first two principles, we believe that the choice of control data should neatly separate the application logic from the adaptation logic, and should be clearly identified and encapsulated in a specific component of a suitable adaptation loop, in order to guarantee an understandable, modular design. For this purpose, sound design principles should be developed in order to ensure correctness-by-design, and guidelines for the development of adaptive systems conforming to well-understood patterns.

As for the third principle (computational reflection), we believe that higher-order forms of control data are to be preferred if computationally affordable, since they make it easy to carry the life-cycle of reliable adaptive systems to runtime, by providing runtime models that can be used to monitor, predict and modify the systems.

In Fig. 11 we recap how the (macro) classes of control data identified in Fig. 1 and discussed in Sections 3–5 (i.e. the rows of the table in Fig. 11) have been exploited for adaptation along the three pillars of Computer Science (i.e. the columns of the table

	Architectures		Models			Languages		
adaptation strategy			4.1				5.2	5.3
architecture	3.1	3.2	4.1		4.3			5.3
entire program			4.1	4.2	4.3			
operation mode			4.1	4.2	4.3	5.1	5.3	4.3

Fig. 11. Control data classes per pillars

Fig. 11) that structure those sections. Broadly speaking, the presence of blank cells in the table suggests us two main interesting and maybe surprising facts, which are concerned with: (i) the use of reflection in programming languages for adaptation; and (ii) the abstraction from operational aspects in architectural approaches to adaptation.

While it is out of doubt that reflection offers a natural mechanism to implement adaptation, our analysis shows that it is more common to allow only some controlled form of reflection in languages designed for programming adaptive systems. This is witnessed by the fact that the class "entire program" has no direct representative in the pillar "Languages". Our understanding is that reflection as-it-is does not offer a convenient abstraction to programmers, because it is too powerful and too risky (i.e. error-prone).

Regarding the pillar "Architectures", it seems that the only class of control data exploited for adaptation is that of "architecture" themselves (e.g. components and their connections), whereas operational aspects are disregarded such as those related to the *how* and *why* questions. While one can argue that both classes "entire program" and "operation mode" of adaptation can somehow be represented at the architecture level (e.g. the notion of component replacement can be instantiated to both such classes), we think that the same does not apply to the class "adaptation strategy". This observation was implicit in [Bradbury et al. 2004], where a lack in meta-levels for the architectural formalisms was already noted. To fill the gap exposed in Fig. 11, we believe that defining an architectural reference model of adaptation that has adaptation strategies as control data would be an interesting subject of further studies.

REFERENCES

ADLER, R., SCHAEFER, I., SCHÜLE, T., AND VECCHIÉ, E. 2007. From model-based design to formal verification of adaptive embedded systems. In *9th International Conference on Formal Engineering Methods (ICFEM 2007)*, M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, Eds. Lecture Notes in Computer Science Series, vol. 4789. Springer, 76–95.

AGHA, G. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA.

ALLEN, R., DOUENCE, R., AND GARLAN, D. 1998. Specifying and analyzing dynamic software architectures. In *1st International Conference on Fundamental Approaches to Software Engineering (FASE 1998)*, E. Astesiano, Ed. Lecture Notes in Computer Science Series, vol. 1382. Springer, 21–37.

ANDERSSON, J., DE LEMOS, R., MALEK, S., AND WEYNS, D. 2009a. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 27–47.

ANDERSSON, J., DE LEMOS, R., MALEK, S., AND WEYNS, D. 2009b. Reflecting on self-adaptive software systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*. IEEE Computer Society, 38–47.

ANDRADE, L. F. AND FIADEIRO, J. L. 2002. An architectural approach to auto-adaptive systems. In *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2002)*. IEEE Computer Society, 439–444.

APPELTAUER, M., HIRSCHFELD, R., HAUPT, M., AND MASUHARA, H. 2011. ContextJ: Context-oriented programming with Java. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software 28,* 1, 272–292.

AUTILI, M., BENEDETTO, P. D., AND INVERARDI, P. 2010. A programming model for adaptable java applications. In *8th International Conference on Principles and Practice of Programming in Java (PPPJ 2010)*, A. Krall and H. Mössenböck, Eds. ACM, 119–128.

BEAL, J., CLEVELAND, J., AND USBECK, K. 2012. Self-stabilizing robot team formation with proto: Ieee self-adaptive and self-organizing systems 2012 demo entry. In *6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012)*. IEEE Computer Society, 233–234.

BIYANI, K. N. AND KULKARNI, S. S. 2008. Assurance of dynamic adaptation in distributed systems. *Journal of Parallel and Distributed Computing 68,* 8, 1097–1112.

BOSCH, J. 1999. Superimposition: a component adaptation technique. *Information & Software Technology 41,* 5, 257–273.

BOUCHACHIA, A. AND NEDJAH, N. 2012. Introduction to the special section on self-adaptive systems: Models and algorithms. *ACM Transactions on Autonomous and Adaptive Systems 7,* 1, 13:1–13:4.

BRACCIALI, A., BROGI, A., AND CANAL, C. 2005. A formal approach to component adaptation. *Journal of Systems and Software 74,* 1, 45–54.

BRADBURY, J. S., CORDY, J. R., DINGEL, J., AND WERMELINGER, M. 2004. A survey of self-management in dynamic software architecture specifications. In *1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, D. Garlan, J. Kramer, and A. L. Wolf, Eds. ACM, 28–33.

BRAVETTI, M., GIUSTO, C. D., PÉREZ, J. A., AND ZAVATTARO, G. 2012. Adaptable processes. *Logical Methods in Computer Science 8,* 4, 13:1–13:71.

BROY, M., LEUXNER, C., SITOU, W., SPANFELNER, B., AND WINTER, S. 2009. Formalizing the notion of adaptive system behavior. In *2009 ACM Symposium on Applied Computing (SAC 2009)*, S. Y. Shin and S. Ossowski, Eds. ACM, 1029–1033.

BRUN, Y., SERUGENDO, G. D. M., GACEK, C., GIESE, H., KIENLE, H. M., LITOIU, M., MÜLLER, H. A., PEZZÈ, M., AND SHAW, M. 2009. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 48–70.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LAFUENTE, A. L., AND VANDIN, A. 2013. Adaptable transition systems. In *21st International Workshop on Algebraic Development Techniques (WADT 2012)*. Lecture Notes in Computer Science Series, vol. 7841. Springer, 95–110.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LLUCH-LAFUENTE, A., AND VANDIN, A. 2012a. A conceptual framework for adaptation. In *15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, J. de Lara and A. Zisman, Eds. Lecture Notes in Computer Science Series, vol. 7212. Springer, 240–254.

BRUNI, R., CORRADINI, A., GADDUCCI, F., LLUCH-LAFUENTE, A., AND VANDIN, A. 2012b. Modelling and analyzing adaptive self-assembly strategies with Maude. In *9th International Workshop on Rewriting Logic and Its Applications (WRLA 2012)*, F. Durán, Ed. Lecture Notes in Computer Science Series, vol. 7571. Springer, 118–138.

BUCCHIARONE, A., CAPPIELLO, C., NITTO, E. D., KAZHAMIAKIN, R., MAZZA, V., AND PISTORE, M. 2010. Design for adaptation of service-based applications: Main issues and requirements. In *2009 International Conference on Service-Oriented Computing (ICSOC/ServiceWave 2009)*, A. Dan, F. Gittler, and F. Toumani, Eds. Lecture Notes in Computer Science Series, vol. 6275. Springer, 467–476.

BUCCHIARONE, A., PISTORE, M., RAIK, H., AND KAZHAMIAKIN, R. 2011. Adaptation of service-based business processes by context-aware replanning. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2011)*, K.-J. Lin, C. Huemer, M. B. Blake, and B. Benatallah, Eds. IEEE Computer Society, 1–8.

CABRI, G., PUVIANI, M., AND ZAMBONELLI, F. 2011. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *2011 International Conference on Collaboration Technologies and Systems (CTS 2011)*, W. W. Smari and G. Fox, Eds. IEEE Computer Society, 508–515.

CÁMARA, J., MARTÍN, J. A., SALAÜN, G., CUBO, J., OUEDERNI, M., CANAL, C., AND PIMENTEL, E. 2009. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In *31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 627–630.

CHENG, B. H. C., DE LEMOS, R., GIESE, H., INVERARDI, P., MAGEE, J., ANDERSSON, J., BECKER, B., BENCOMO, N., BRUN, Y., CUKIC, B., SERUGENDO, G. D. M., DUSTDAR, S., FINKELSTEIN, A., GACEK, C., GEIHS, K., GRASSI, V., KARSAI, G., KIENLE, H. M., KRAMER, J., LITOIU, M., MALEK, S., MIRANDOLA, R., MÜLLER, H. A., PARK, S., SHAW, M., TICHY, M., TIVOLI, M., WEYNS, D., AND WHITTLE, J. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Lecture Notes in Computer Science Series, vol. 5525. Springer, 1–26.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND TALCOTT, C. L. 2007. *All About Maude*. Lecture Notes in Computer Science Series, vol. 4350. Springer.

CLEMENTS, P. 1996. A survey of architecture description languages. In *8th International Workshop on Software Specification and Design (IWSSD 1996)*. IEEE Computer Society, 16–25.

CORDY, M., CLASSEN, A., HEYMANS, P., SCHOBBENS, P.-Y., AND LEGAY, A. 2013. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems*, J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds. Springer, 1–29.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *8th European Software Engineering Conference/9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2001)*. ACM SIGSOFT Software Engineering Notes Series, vol. 26(5). ACM, 109–120.

DE LEMOS, R., GIESE, H., MÜLLER, H., SHAW, M., ANDERSSON, J., BARESI, L., BECKER, B., BENCOMO, N., BRUN, Y., CIKIC, B., DESMARAIS, R., DUSTDAR, S., ENGELS, G., GEIHS, K., GOESCHKA, K. M., GORLA, A., GRASSI, V., INVERARDI, P., KARSAI, G., KRAMER, J., LITOIU, M., LOPES, A., MAGEE, J., MALEK, S., MANKOVSKII, S., MIRANDOLA, R., MYLOPOULOS, J., NIERSTRASZ, O., PEZZÈ, M., PREHOFER, C., SCHÄFER, W., SCHLICHTING, W., SCHMERL, B., SMITH, D. B., SOUSA, J. P., TAMURA, G., TAHVILDARI, L., VILLEGAS, N. M., VOGEL, T., WEYNS, D., WONG, K., AND WUTTKE, J. 2011. Software engineering for self-adpaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems*, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds. Number 10431 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

DE NICOLA, R., FERRARI, G., LORETI, M., AND PUGLIESE, R. 2013. A language-based approach to autonomic computing. In *10th International Symposium on Formal Methods for Components and Objects (FMCO 2011)*, B. Beckert, F. Damiani, F. S. de Boer, and M. M. Bonsangue, Eds. Lecture Notes in Computer Science Series, vol. 7542. Springer, 25–48.

DE NICOLA, R., FERRARI, G. L., AND PUGLIESE, R. 1998. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering 24,* 5, 315–330.

DOBSON, S., DENAZIS, S. G., FERNÁNDEZ, A., GAÏTI, D., GELENBE, E., MASSACCI, F., NIXON, P., SAFFRE, F., SCHMIDT, N., AND ZAMBONELLI, F. 2006. A survey of autonomic communications. *TAAS 1,* 2, 223–259.

DOWLING, J., SCHÄFER, T., CAHILL, V., HARASZTI, P., AND REDMOND, B. 2000. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *1st OOPSLA Workshop on Reflection and Software Engineering*, W. Cazzola, R. J. Stroud, and F. Tisato, Eds. Lecture Notes in Computer Science Series, vol. 1826. Springer, 169–188.

ECKHARDT, J., MÜHLBAUER, T., MESEGUER, J., AND WIRSING, M. 2013. Statistical model-checking for composite actor systems. In *21st International Workshop on Algebraic Development Techniques (WADT 2012)*. Lecture Notes in Computer Science Series, vol. 7841. Springer, 143–160.

EHRIG, H., ERMEL, C., RUNGE, O., BUCCHIARONE, A., AND PELLICCIONE, P. 2010. Formal analysis and verification of self-healing systems. In *13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, D. S. Rosenblum and G. Taentzer, Eds. Lecture Notes in Computer Science Series, vol. 6013. Springer, 139–153.

FOURNET, C. AND GONTHIER, G. 2002. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics (APPSEM 2000)*, G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds. Lecture Notes in Computer Science Series, vol. 2395. Springer, 268–332.

FRITSCH, S., SENART, A., SCHMIDT, D. C., AND CLARKE, S. 2008. Time-bounded adaptation for automotive system software. In *30th International Conference on Software Engineering (ICSE 2008)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 571–580.

GHEZZI, C., PRADELLA, M., AND SALVANESCHI, G. 2011. An evaluation of the adaptation capabilities in programming languages. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011)*, H. Giese and B. H. C. Cheng, Eds. ACM, 50–59.

GJONDREKAJ, E., LORETI, M., PUGLIESE, R., AND TIEZZI, F. 2012. Modeling adaptation with a tuple-based coordination language. In *2012 ACM Symposium on Applied Computing (SAC 2012)*, S. Ossowski and P. Lecca, Eds. ACM, 1522–1527.

GREENWOOD, P. AND BLAIR., L. 2004. Using dynamic aspect-oriented programming to implement an autonomic system. In *2004 Dynamic Aspects Workshop (DAW 2004)*. RIACS, 76–88.

HARVEY, I., PAOLO, E. A. D., WOOD, R., QUINN, M., AND TUCI, E. 2005. Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life 11,* 1-2, 79–98.

HINCHEY, M. G. AND STERRITT, R. 2006. Self-managing software. *IEEE Computer 39,* 2, 107–109.

HIRSCHFELD, R., COSTANZA, P., AND NIERSTRASZ, O. 2008. Context-oriented programming. *Journal of Object Technology 7,* 3, 125–151.

HÖLZL, M. M. AND WIRSING, M. 2011. Towards a system model for ensembles. In *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy, and J. Meseguer, Eds. Lectures Notes in Computer Science Series, vol. 7000. Springer, 241–261.

HORN, P. 2001. Autonomic computing: IBM's perspective on the state of information technology.

HUEBSCHER, M. C. AND MCCANN, J. A. 2008. A survey of autonomic computing - degrees, models, and applications. *ACM Computing Survey 40,* 3, 7:1–7:28.

IBM CORPORATION. 2005. An architectural blueprint for autonomic computing. Tech. rep., IBM.

IFTIKHAR, M. U. AND WEYNS, D. 2012. A case study on formal verification of self-adaptive behaviors in a decentralized system. In *11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012)*, N. Kokash and A. Ravara, Eds. Electronics Proceedings in Theoretical Computer Science Series, vol. 91. EPTCS, 45–62.

KAPITSAKI, G. M., PREZERAKOS, G. N., TSELIKAS, N. D., AND VENIERIS, I. S. 2009. Context-aware service engineering: A survey. *Journal of Systems and Software 82,* 8, 1285–1297.

KARSAI, G. AND SZTIPANOVITS, J. 1999. A model-based approach to self-adaptive software. *Intelligent Systems and their Applications 14,* 3, 46–53.

KHAKPOUR, N., JALILI, S., TALCOTT, C., SIRJANI, M., AND MOUSAVI, M. 2012. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming 78,* 1, 3 – 26.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., MARC LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP 1997)*, M. Aksit and S. Matsuoka, Eds. Lecture Notes in Computer Science Series, vol. 1241. Springer, 220–242.

KOUNEV, S. 2011. Self-aware software and systems engineering: A vision and research roadmap. *GI Softwaretechnik-Trends 31,* 4, 1–5.

KRAMER, J. AND MAGEE, J. 2009. A rigorous architectural approach to adaptive software engineering. *J. Comput. Sci. Technol. 24,* 2, 183–188.

LADDAGA, R. 1997. Self-adaptive software. Tech. Rep. 98-12, DARPA BAA.

LANESE, I., BUCCHIARONE, A., AND MONTESI, F. 2010. A framework for rule-based dynamic adaptation. In *5th International Conference on Trustworthly Global Computing (TGC 2010)*, M. Wirsing, M. Hofmann, and A. Rauschmayer, Eds. Lecture Notes in Computer Science Series, vol. 6084. Springer, 284–300.

LINTS, T. 2010. The essentials in defining adaptation. In *4th Annual IEEE Systems Conference*. IEEE Computer Society, 113–116.

MARANINCHI, F. AND RÉMOND, Y. 1998. Mode-automata: About modes and states for reactive systems. In *7th European Symposium on Programming (ESOP 1998)*, C. Hankin, Ed. Lecture Notes in Computer Science Series, vol. 1381. Springer, 185–199.

MARTÍN, J. A., BROGI, A., AND PIMENTEL, E. 2012. Learning from failures: A lightweight approach to run-time behavioural adaptation. In *8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, F. Arbab and P. C. Ölveczky, Eds. Lecture Notes in Computer Science Series, vol. 7253. Springer, 259–277.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., AND CHENG, B. H. C. 2004. Composing adaptive software. *IEEE Computer 37,* 7, 56–64.

MERELLI, E., PAOLETTI, N., AND TESEI, L. 2012. A multi-level model for self-adaptive systems. In *11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA 2012)*, N. Kokash and A. Ravara, Eds. Electronics Proceedings in Theoretical Computer Science Series, vol. 91. EPTCS, 112–126.

MESEGUER, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96,* 1, 73–155.

MESEGUER, J. AND TALCOTT, C. 2002. Semantic models for distributed object reflection. In *16th European Conference on Object-Oriented Programming (ECOOP 2002)*, B. Magnusson, Ed. Lecture Notes in Computer Science Series, vol. 2374. Springer, 1–36.

MIKIC-RAKIC, M. AND MEDVIDOVIC, N. 2006. A classification of disconnected operation techniques. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2006)*. IEEE, 144–151.

MILNER, R. 1999. *Communicating and mobile systems - the π-calculus*. Cambridge University Press.

MONTESI, F., GUIDI, C., LUCCHI, R., AND ZAVATTARO, G. 2007. JOLIE: a Java orchestration language interpreter engine. In *2nd International Workshop on Coordination and Organization (CoOrg 2006) and 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006)*, G. Boella, M. Dastani, A. Omicini, L. W. van der Torre, I. Cerna, and I. Linden, Eds. Electronics Notes in Theoretical Computer Science Series, vol. 181. Elsevier, 19–33.

MÜHL, G., WERNER, M., JAEGER, M., HERRMANN, K., AND PARZYJEGLA, H. 2007. On the definitions of self-managing and self-organizing systems. In *2007 ITG-GI Conference on Communication in Distributed Systems (KiVS)*. IEEE Computer Society.

O'GRADY, R., GROSS, R., CHRISTENSEN, A. L., AND DORIGO, M. 2010. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots 28,* 4, 439–455.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S., AND WOLF, A. L. 1999. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications 14,* 3, 54–62.

PAVLOVIC, D. 2000. Towards semantics of self-adaptive software. In *First International Workshop on Self-Adaptive Software (IWSAS 2000)*, P. Robertson, H. E. Shrobe, and R. Laddaga, Eds. Lecture Notes in Computer Science Series, vol. 1936. Springer, 65–74.

POPESCU, R., STAIKOPOULOS, A., BROGI, A., LIU, P., AND CLARKE, S. 2012. A formalized, taxonomy-driven approach to cross-layer application adaptation. *ACM Transactions on Autonomous and Adaptive Systems 7,* 1, 7:1–7:30.

POPOVICI, A., ALONSO, G., AND GROSS, T. R. 2003. Just-in-time aspects: efficient dynamic weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*. ACM, 100–109.

PUKALL, M., KÄSTNER, C., CAZZOLA, W., GÖTZ, S., GREBHAHN, A., SCHRÖTER, R., AND SAAKE, G. 2013. Javadaptor - flexible runtime updates of Java applications. *Software, Practice and Experience 43,* 2, 153–185.

RAIBULET, C. 2008. Facets of adaptivity. In *Second European Conference on Software Architecture (ECSA 2008)*, R. Morrison, D. Balasubramaniam, and K. E. Falkner, Eds. Lecture Notes in Computer Science Series, vol. 5292. Springer, 342–345.

SAGASTI, F. 1970. A conceptual and taxonomic framework for the analysis of adaptive behavior. *General Systems XV*, 151–160.

SALEHIE, M. AND TAHVILDARI, L. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems 4,* 2, 14:1–14:42.

SALVANESCHI, G., GHEZZI, C., AND PRADELLA, M. 2011. Context-oriented programming: A programming paradigm for autonomic systems. Tech. Rep. abs/1105.0069, CoRR.

SALVANESCHI, G., GHEZZI, C., AND PRADELLA, M. 2013. Towards language-level support for self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems*. to appear.

SANGIORGI, D. 1992. Expressing mobility in process algebras: First-order and higher-order paradigms. Ph.D. thesis, University of Edinburgh.

SCHAEFER, I. AND POETZSCH-HEFFTER, A. 2006. Using abstraction in modular verification of synchronous adaptive systems. In *Workshop on "Trustworthy Software"*, S. Autexier, S. Merz, L. W. N. van der Torre, R. Wilhelm, and P. Wolper, Eds. OASICS Series, vol. 3. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

TALCOTT, C. L. 2006. Coordination models based on a formal model of distributed object reflection. In *First International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005)*, L. Brim and I. Linden, Eds. Electronics Notes in Theoretical Computer Science Series, vol. 150(1). Elsevier, 143–157.

TALCOTT, C. L. 2007. Policy-based coordination in PAGODA: A case study. In *2nd International Workshop on Coordination and Organization (CoOrg 2006) and 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006)*, G. Boella, M. Dastani, A. Omicini, L. W. van der Torre, I. Cerna, and I. Linden, Eds. Electronics Notes in Theoretical Computer Science Series, vol. 181. Elsevier, 97–112.

VAN RENESSE, R., BIRMAN, K. P., HAYDEN, M., VAYSBURD, A., AND KARR, D. A. 1998. Building adaptive systems using ensemble. *Software, Practice and Experience 28,* 9, 963–979.

VIROLI, M., CASADEI, M., MONTAGNA, S., AND ZAMBONELLI, F. 2011. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems 6,* 2, 14:1–14:24.

WANG, H., LV, H., AND FENG, G. 2009. A self-reflection model for autonomic computing systems based on π-calculus. In *3rd International Conference on Network and System Security (NSS 2009)*, Y. Xiang, J. Lopez, H. Wang, and W. Zhou, Eds. IEEE Computer Society, 310–315.

WEYNS, D., MALEK, S., AND ANDERSSON, J. 2012. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems 7,* 1, 8:1–8:61.

WIRTH, N. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall.

ZADEH, L. A. 1963. On the definition of adaptivity. *Proceedings of the IEEE 3,* 51, 469–470.

ZHANG, J. AND CHENG, B. H. C. 2006a. Model-based development of dynamically adaptive software. In *28th International Conference on Software Engineering (ICSE 2006)*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 371–380.

ZHANG, J. AND CHENG, B. H. C. 2006b. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software 79,* 10, 1361–1369.

ZHANG, J., GOLDSBY, H., AND CHENG, B. H. C. 2009. Modular verification of dynamically adaptive systems. In *8th International Conference on Aspect-Oriented Software Development (AOSD 2009)*, K. J. Sullivan, A. Moreira, C. Schwanninger, and J. Gray, Eds. ACM, 161–172.

ZHAO, Y., MA, D., LI, J., AND LI, Z. 2011. Model checking of adaptive programs with mode-extended linear temporal logic. In *8th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe 2011)*. IEEE Computer Society, 40–48.